

statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1: 10 K=10  
20 FOR I=1 TO K STEP 2  
30 PRINT I;  
40 K=K+10  
50 PRINT K  
60 NEXT  
RUN  
1 20  
3 30  
5 40  
7 50  
9 60  
Ok

Example 2: 10 J=0  
20 FOR I=1 TO J  
30 PRINT I  
40 NEXT I

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3: 10 I=5  
20 FOR I=1 TO I+5  
30 PRINT I;  
40 NEXT  
RUN  
1 2 3 4 5 6 7 8 9 10  
Ok

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set. (Note: Previous versions of Microsoft BASIC set the initial value of the loop variable before setting the final value; i.e., the above loop would have executed six times.)

### 2.23 GET

**Format:** GET [#]<file number>[,<record number>]

**Purpose:** To read a record from a random disk file into a random buffer.

**Remarks:** <file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

**Example:** See Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

**NOTE:** After a GET statement, INPUT# and LINE INPUT# may be done to read characters from the random file buffer.

## 2.24 GOSUB...RETURN

Format:       GOSUB <line number>

          .  
          .  
          .

          RETURN

Purpose:       To branch to and return from a subroutine.

Remarks:     <line number> is the first line of the  
              subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertant entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

Example:      10 GOSUB 40  
              20 PRINT "BACK FROM SUBROUTINE"  
              30 END  
              40 PRINT "SUBROUTINE";  
              50 PRINT " IN";  
              60 PRINT " PROGRESS"  
              70 RETURN  
              RUN  
              SUBROUTINE IN PROGRESS  
              BACK FROM SUBROUTINE  
              Ok

2.25 GOTO

Format: GOTO <line number>

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example: LIST  
10 READ R  
20 PRINT "R =";R,  
30 A = 3.14\*R^2  
40 PRINT "AREA =";A  
50 GOTO 10  
60 DATA 5,7,12  
Ok  
RUN  
R = 5                    AREA = 78.5  
R = 7                    AREA = 153.86  
R = 12                   AREA = 452.16  
?Out of data in 10  
Ok

## 2.26 IF...THEN[...ELSE] AND IF...GOTO

Format: IF <expression> THEN <statement(s)> | <line number>  
[ELSE <statement(s)> | <line number>]

Format: IF <expression> GOTO <line number>  
[ELSE <statement(s)> | <line number>]

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma is allowed before THEN.

### Nesting of IF Statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X  
    THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example

```
IF A=B THEN IF B=C THEN PRINT "A=C"  
    ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in the indirect mode.

NOTE: When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1: 200 IF I THEN GET#1,I

This statement GETs record number I if I is not zero.

Example 2: 100 IF(I<20)\*(I>10) THEN DB=1979-1:GOTO 300  
110 PRINT "OUT OF RANGE"

.  
.  
.

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3: 210 IF IOFLAG THEN PRINT A\$ ELSE LPRINT A\$

This statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

## 2.27 INPUT

**Format:** INPUT[;][<"prompt string">]<list of variables>

**Purpose:** To allow input from the terminal during program execution.

**Remarks:** When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B\$ will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not echo a carriage return/line feed sequence.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

Examples: 10 INPUT X  
20 PRINT X "SQUARED IS" X^2  
30 END  
RUN  
? 5 (The 5 was typed in by the user  
in response to the question mark.)  
5 SQUARED IS 25  
Ok

LIST  
10 PI=3.14  
20 INPUT "WHAT IS THE RADIUS";R  
30 A=PI\*R^2  
40 PRINT "THE AREA OF THE CIRCLE IS";A  
50 PRINT  
60 GOTO 20  
Ok  
RUN  
WHAT IS THE RADIUS? 7.4 (User types 7.4)  
THE AREA OF THE CIRCLE IS 171.946  
  
WHAT IS THE RADIUS?  
etc.

## 2.28 INPUT#

**Format:** INPUT#<file number>,<variable list>

**Purpose:** To read data items from a sequential disk file and assign them to program variables.

**Remarks:** <file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma.

If BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

**Example:** See Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

2.29 KILL

Format: KILL <filename>

Purpose: To delete a file from disk.

Remarks: If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

KILL is used for all types of disk files: program files, random data files and sequential data files.

Example: 200 KILL "DATA1"

See also PART II, Chapter 3, Microsoft BASIC Disk I/O, of the Microsoft BASIC User's Guide.

2.30 LET

Format:       [LET] <variable>=<expression>

Purpose:       To assign the value of an expression to a variable.

Remarks:     Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:      110 LET D=12  
              120 LET E=12^2  
              130 LET F=12^4  
              140 LET SUM=D+E+F

      .  
      .  
      .

      or

              110 D=12  
              120 E=12^2  
              130 F=12^4 ,  
              140 SUM=D+E+F

      .  
      .  
      .

### 2.31 LINE INPUT

**Format:** LINE INPUT[;][<"prompt string">];<string variable>

**Purpose:** To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

**Remarks:** The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to <string variable>. However, if a line feed/carriage return sequence (this order only) is encountered, both characters are echoed; but the carriage return is ignored, the line feed is put into <string variable>, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be escaped by typing Control-C. BASIC will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT.

**Example:** See Example, Section 2.32, LINE INPUT#.

### 2.32 LINE INPUT#

Format: LINE INPUT#<file number>,<string variable>

Purpose: To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks: <file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

Example:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES      234,4      MEMPHIS
LINDA JONES      234,4      MEMPHIS
Ok
```

### 2.33 LIST

Format 1: LIST [<line number>]

Format 2: LIST [<line number>[-<line number>]]

Purpose: To list all or part of the program currently in memory at the terminal.

Remarks: BASIC always returns to command level after a LIST is executed.

Format 1: If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by typing Control-C.) If <line number> is included, only the specified line will be listed.

Format 2: This format allows the following options:

1. If only the first number is specified, that line and all higher-numbered lines are listed.
2. If only the second number is specified, all lines from the beginning of the program through that line are listed.
3. If both numbers are specified, the entire range is listed.

Examples:   Format 1:

LIST               Lists the program currently  
                    in memory.

LIST 500           Lists line 500.

Format 2:

LIST 150-          Lists all lines from 150  
                    to the end.

LIST -1000          Lists all lines from the  
                    lowest number through 1000.

LIST 150-1000      Lists lines 150 through  
                    1000, inclusive.

2.34 LLIST

Format: LLIST [<line number>[-<line number>]]

Purpose: To list all or part of the program currently in memory at the line printer.

Remarks: LLIST assumes a 132-character wide printer.

BASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Format 2.

NOTE: LLIST and LPRINT are not included in all implementations of Microsoft BASIC.

Example: See the examples for LIST, Format 2.

### 2.35 LOAD

**Format:** LOAD <filename>[,R]

**Purpose:** To load a file from disk into memory.

**Remarks:** <filename> is the name that was used when the file was SAVED. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to the Microsoft BASIC User's Guide, PART III for information about possible filename extensions under your operating system.)

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the "R" option is used with LOAD, the program is RUN after it is LOADED, and all open data files are kept open. Thus, LOAD with the "R" option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

**Example:** LOAD "STRTRK",R

2.36 LPRINT AND LPRINT USING

Format: LPRINT [<list of expressions>]

LPRINT USING <string exp>;<list of expressions>

Purpose: To print data at the line printer.

Remarks: Same as PRINT and PRINT USING, except output goes to the line printer. See Section 2.49 and Section 2.50.

LPRINT assumes a 132-character-wide printer.

NOTE: LPRINT and LLIST are not included in all implementations of Microsoft BASIC.

### 2.37 LSET AND RSET

**Format:** LSET <string variable> = <string expression>  
RSET <string variable> = <string expression>

**Purpose:** To move data from memory to a random file buffer (in preparation for a PUT statement).

**Remarks:** If <string expression> requires fewer bytes than were FIELDed to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See the MKI\$, MKS\$, MKD\$ functions, Section 3.25.

**Examples:** 150 LSET A\$=MKS\$(AMT)  
160 LSET D\$=DESC(\$)

See also PART II, Chapter 3, Microsoft BASIC Disk I/O, of the Microsoft BASIC User's Guide.

**NOTE:** LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N\$ in a 20-character field. This can be very handy for formatting printed output.

### 2.38 MERGE

Format:       MERGE <filename>

Purpose:       To merge a specified disk file into the program currently in memory.

Remarks:     <filename> is the name used when the file was SAVED. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to the Microsoft BASIC User's Guide, PART III for information about possible filename extensions under your operating system.) The file must have been SAVED in ASCII format. (If not, a "Bad file mode" error occurs.)

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disk into the program in memory.)

BASIC always returns to command level after executing a MERGE command.

Example:      MERGE "NUMBRS"

### 2.39 MID\$

Format:        MID\$(<string exp1>,n[,m])=<string exp2>

where n and m are integer expressions and  
<string exp1> and <string exp2> are string  
expressions.

Purpose:        To replace a portion of one string with another  
string.

Remarks:      The characters in <string exp1>, beginning at  
position n, are replaced by the characters in  
<string exp2>. The optional m refers to the  
number of characters from <string exp2> that  
will be used in the replacement. If m is  
omitted, all of <string exp2> is used. However,  
regardless of whether m is omitted or included,  
the replacement of characters never goes beyond  
the original length of <string exp1>.

Example:       10 A\$="KANSAS CITY, MO"  
                 20 MID\$(A\$,14)="KS"  
                 30 PRINT A\$  
                 RUN  
                 KANSAS CITY, KS

MID\$ is also a function that returns a substring  
of a given string. See Section 3.24.

2.40 NAME

Format:       NAME <old filename> AS <new filename>

Purpose:       To change the name of a disk file.

Remarks:     <old filename> must exist and <new filename>  
must not exist; otherwise an error will result.  
After a NAME command, the file exists on the  
same disk, in the same area of disk space, with  
the new name.

Example:      Ok  
              NAME "ACCTS" AS "LEDGER"  
              Ok

In this example, the file that was  
formerly named ACCTS will now be named LEDGER.

2.41 NEW

Format: NEW

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC always returns to command level after a NEW is executed.

2.42 NULL

Format: NULL <integer expression>

Purpose: To set the number of nulls to be printed at the end of each line.

Remarks: For 10-character-per-second tape punches, <integer expression> should be >=3. When tapes are not being punched, <integer expression> should be 0 or 1 for Teletypes and Teletype-compatible terminal screens. <integer expression> should be 2 or 3 for 30 cps hard copy printers. The default value is 0.

Example: Ok  
NULL 2  
Ok  
100 INPUT X  
200 IF X<50 GOTO 800

.  
.  
.

Two null characters will be printed after each line.

### 2.43 ON ERROR GOTO

**Format:** ON ERROR GOTO <line number>

**Purpose:** To enable error trapping and specify the first line of the error handling subroutine.

**Remarks:** Once error trapping has been enabled all errors detected, including direct mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine. If <line number> does not exist, an "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

**NOTE:** If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

**Example:** 10 ON ERROR GOTO 1000

2.44 ON...GOSUB AND ON...GOTO

Format:       ON <expression> GOTO <list of line numbers>  
              ON <expression> GOSUB <list of line numbers>

Purpose:       To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks:     The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

Example:      100 ON L-1 GOTO 150,300,320,390

2.45 OPEN

Format: OPEN <mode>,[#]<file number>,<filename>,[<reclen>]

Purpose: To allow I/O to a disk file.

Remarks: A disk file must be OPENed before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose first character is one of the following:

O specifies sequential output mode

I specifies sequential input mode

R specifies random input/output mode

<file number> is an integer expression whose value is between one and fifteen. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.

<filename> is a string expression containing a name that conforms to your operating system's rules for disk filenames.

<reclen> is an integer expression which, if included, sets the record length for random files. The default record length is 128 bytes.

NOTE: A file can be OPENed for sequential input or random access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

Example: 10 OPEN "I",2,"INVEN"

See also PART II, Chapter 3, Microsoft BASIC Disk I/O, of the Microsoft BASIC User's Guide.

2.46 OPTION BASE

Format:       OPTION BASE n  
              where n is 1 or 0

Purpose:       To declare the minimum value for array  
              subscripts.

Remarks:     The default base is 0. If the statement

              OPTION BASE 1

is executed, the lowest value an array subscript  
may have is one.

2.47 OUT

**Format:**       OUT I,J  
                  where I and J are integer expressions in the  
                  range 0 to 255.

**Purpose:**       To send a byte to a machine output port.

**Remarks:**     The integer expression I is the port number, and  
                  the integer expression J is the data to be  
                  transmitted.

**Example:**      100 OUT 32,100

2.48 POKE

Format: POKE I,J  
where I and J are integer expressions

Purpose: To write a byte into a memory location.

Remarks: The integer expression I is the address of the memory location to be POKEd. The integer expression J is the data to be POKEd. J must be in the range 0 to 255. I must be in the range 0 to 65536.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Section 3.27.

POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example: 10 POKE &H5A00,&HFF

## 2.49 PRINT

Format: PRINT [<list of expressions>]

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

### Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8(-7) is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-15 is output as .0000000000000001 and 1D-16 is output as 1D-16.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1: 10 X=5  
20 PRINT X+5, X-5, X\*(-5), X^5  
30 END  
RUN  
10                    0                    -25                    3125  
Ok

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2: LIST  
10 INPUT X  
20 PRINT X "SQUARED IS" X^2 "AND";  
30 PRINT X "CUBED IS" X^3  
40 PRINT  
50 GOTO 10  
Ok  
RUN  
? 9  
9 SQUARED IS 81 AND 9 CUBED IS 729  
  
? 21  
21 SQUARED IS 441 AND 21 CUBED IS 9261  
  
?

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

Example 3: 10 FOR X = 1 TO 5  
20 J=J+5  
30 K=K+10  
40 ?J;K;  
50 NEXT X  
Ok  
RUN  
5 10 10 20 15 30 20 40 25 50  
Ok

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

2.50 PRINT USING

Format: PRINT USING <string exp>;<list of expressions>

Purpose: To print strings or numbers using a specified format.

Remarks <list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <string exp> is a string literal (or variable) comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

Examples: be printed, separated by semicolons. <string exp> is a string literal (or variable) comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!" Specifies that only the first character in the given string is to be printed.

"\n spaces\" Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right. Example:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\ \";A$;B$
50 PRINT USING "\ \ \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input. Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

# A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

. A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
0.78
```

```
PRINT USING "###.##";987.654
987.65
```

```
PRINT USING "##.##  ";10.2,5.3,66.789,.234
10.20   5.30   66.79   0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+ A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

- A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##  ";-68.95,2.4,55.6,-.9
-68.95   +2.40   +55.60   -0.90
```

```
PRINT USING "##.##-  ";-68.95,22.449,-7.01
68.95-   22.45    7.01-
```

\*\* A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The \*\* also specifies positions for two more digits.

```
PRINT USING "***.##  ";12.39,-0.9,765.1
*12.4   *-0.9   765.1
```

**\$\$** A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The **\$\$** specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with **\$\$**. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
$456.78
```

**\*\*\$** The **\*\*\$** at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **\*\*\$** specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$###.##";2.34
***$2.34
```

**,** A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (**\_\_**) format.

```
PRINT USING "####,.##";1234.5
1,234.50
```

```
PRINT USING "####.##,";1234.5
1234.50,
```

**^^^^** Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
2.35E+02
```

```
PRINT USING ".####^^^^-";888888
.8889E+06
```

```
PRINT USING "+.##^^^^";123
+.12E+03
```

- An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "\_" in the format string.

- % If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22
```

```
PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

2.51 PRINT# AND PRINT# USING

Format: PRINT#<filenumber>,[USING<string exp>;]<list of exps>

Purpose: To write data to a sequential disk file.

Remarks: <file number> is the number used when the file was OPENed for output. <string exp> is comprised of formatting characters as described in Section 2.50, PRINT USING. The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal screen with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it will be input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A\$="CAMERA" and B\$="93604-1". The statement

```
PRINT#1,A$;B$
```

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;"",";B$
```

The image written to disk is

```
CAMERA,93604-1
```

which can be read back into two string

variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, CHR\$(34).

For example, let A\$="CAMERA, AUTOMATIC" and B\$=" 93604-1". The statement

```
PRINT#1,A$;B$
```

would write the following image to disk:

```
CAMERA, AUTOMATIC 93604-1
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA" to A\$ and "AUTOMATIC 93604-1" to B\$. To separate these strings properly on the disk, write double quotes to the disk image using CHR\$(34). The statement

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disk:

```
"CAMERA, AUTOMATIC" 93604-1"
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A\$ and " 93604-1" to B\$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

For more examples using PRINT#, see Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

See also WRITE#, Section 2.68.

2.52 PUT

Format: PUT [#]<file number>[,<record number>]

Purpose: To write a record from a random buffer to a random disk file.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the record will have the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

Example: See Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

NOTE: PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before a PUT statement.

In the case of WRITE#, BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

### 2.53 RANDOMIZE

**Format:** RANDOMIZE [<expression>]

**Purpose:** To reseed the random number generator.

**Remarks:** If <expression> is omitted, BASIC suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

**Example:**

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)? 3 (user
types 3)
.88598 .484668 .586328 .119426 .709225
Ok
RUN
Random Number Seed (-32768 to 32767)? 4 (user
types 4 for new sequence)
.803506 .162462 .929364 .292443 .322921
Ok
RUN
Random Number Seed (-32768 to 32767)? 3 (same
sequence as first RUN)
.88598 .484668 .586328 .119426 .709225
Ok
```

## 2.54 READ

**Format:** READ <list of variables>

**Purpose:** To read values from a DATA statement and assign them to variables. (See DATA, Section 2.10.)

**Remarks:** A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see RESTORE, Section 2.57)

**Example 1:**

```
.  
.   
.   
80 FOR I=1 TO 10  
90 READ A(I)  
100 NEXT I  
110 DATA 3.08,5.19,3.12,3.98,4.24  
120 DATA 5.08,5.55,4.00,3.16,3.37  
.   
.   
. 
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2: LIST  
10 PRINT "CITY", "STATE", " ZIP"  
20 READ C\$,S\$,Z  
30 DATA "DENVER,", COLORADO, 80211  
40 PRINT C\$,S\$,Z  
Ok  
RUN  
CITY STATE ZIP  
DENVER, COLORADO 80211  
Ok

This program READs string and numeric data from the DATA statement in line 30.

2.55 REM

Format: REM <remark>

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM.

WARNING: Do not use this in a data statement as it would be considered legal data.

Example:

```
.  
.   
.   
120 REM CALCULATE AVERAGE VELOCITY  
130 FOR I=1 TO 20  
140 SUM=SUM + V(I)
```

```
.  
.   
.   

```

or:

```
.  
.   
.   
120 FOR I=1 TO 20      ^CALCULATE AVERAGE VELOCITY  
130 SUM=SUM+V(I)  
140 NEXT I
```

```
.  
.   
.   

```

2.56 RENUM

Format: RENUM [[<new number>][,[<old number>][,<increment>]]]

Purpose: To renumber program lines.

Remarks: <new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyy may be changed.

NOTE: RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

Examples:	RENUM	Renumbers the entire program. The first new line number will be 10. Lines will increment by 10.
	RENUM 300,,50	Renumbers the entire program. The first new line number will be 300. Lines will increment by 50.
	RENUM 1000,900,20	Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

2.57 RESTORE

Format:       RESTORE [<line number>]

Purpose:       To allow DATA statements to be reread from a specified line.

Remarks:     After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:      10 READ A,B,C  
              20 RESTORE  
              30 READ D,E,F  
              40 DATA 57, 68, 79  
              .  
              .  
              .

2.58 RESUME

Formats: RESUME

RESUME 0

RESUME NEXT

RESUME <line number>

Purpose: To continue program execution after an error recovery procedure has been performed.

Remarks: Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME	Execution resumes at the
or	statement which caused the
RESUME 0	error.

RESUME NEXT	Execution resumes at the
	statement immediately fol-
	lowing the one which
	caused the error.

RESUME <line number>	Execution resumes at
	<line number>.

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

Example: 10 ON ERROR GOTO 900

```
.  
. 900 IF (ERR=230)AND(ERL=90) THEN PRINT "TRY  
AGAIN":RESUME 80  
.  .  
.  .  
.  .
```

## 2.59 RUN

Format 1: RUN [<line number>]

Purpose: To execute the program currently in memory.

Remarks: If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC always returns to command level after a RUN is executed.

Example: RUN

Format 2: RUN <filename>[,R]

Purpose: To load a file from disk into memory and run it.

Remarks: <filename> is the name used when the file was SAVED. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to the Microsoft BASIC User's Guide, PART III for information about possible filename extensions under your operating system.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain OPEN.

Example: RUN "NEWFIL",R

See also PART II, Chapter 3, Microsoft BASIC Disk I/O, of the Microsoft BASIC User's Guide.

NOTE: The Microsoft BASIC Compiler supports the RUN and RUN <line number> forms of the RUN statement. The Microsoft BASIC Compiler does not support the "R" option with RUN. If you want this feature, the CHAIN statement should be used.

2.60 SAVE

Format:       SAVE <filename>[,A | ,P]

Purpose:       To save a program file on disk.

Remarks:     <filename> is a quoted string that conforms to your operating system's requirements for filenames. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to the Microsoft BASIC User's Guide, PART III for information about possible filename extensions under your operating system.) If <filename> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOAded), any attempt to list or edit it will fail.

Examples:     SAVE"COM2",A  
              SAVE"PROG",P

See also PART II, Chapter 3, Microsoft BASIC Disk I/O, of the Microsoft BASIC User's Guide.

2.61 STOP

Format: STOP

Purpose: To terminate program execution and return to command level.

Remarks: STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

Break in line nnnnn

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Section 2.8).

Example: 10 INPUT A,B,C  
20  $K=A^2*5.3$ : $L=B^3/.26$   
30 STOP  
40  $M=C*K+100$ :PRINT M  
RUN  
? 1,2,3  
BREAK IN 30  
Ok  
PRINT L  
30.7692  
Ok  
CONT  
115.9  
Ok

2.62 SWAP

Format: SWAP <variable>,<variable>

Purpose: To exchange the values of two variables.

Remarks: Any type variable may be SWAPped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example: LIST  
10 A\$=" ONE " : B\$=" ALL " : C\$="FOR"  
20 PRINT A\$ C\$ B\$  
30 SWAP A\$, B\$  
40 PRINT A\$ C\$ B\$  
RUN  
Ok  
ONE FOR ALL  
ALL FOR ONE  
Ok

### 2.63 TRON/TROFF

Format: TRON

TROFF

Purpose: To trace the execution of program statements.

Remarks: As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example: TRON  
Ok  
LIST  
10 K=10  
20 FOR J=1 TO 2  
30 L=K + 10  
40 PRINT J;K;L  
50 K=K+10  
60 NEXT  
70 END  
Ok  
RUN  
[10][20][30][40] 1 10 20  
[50][60][30][40] 2 20 30  
[50][60][70]  
Ok  
TROFF  
Ok

2.64 WAIT

Format:        WAIT <port number>, I[,J]  
              where I and J are integer expressions

Purpose:        To suspend program execution while monitoring  
              the status of a machine input port.

Remarks:     The WAIT statement causes execution to be  
              suspended until a specified machine input port  
              develops a specified bit pattern. The data read  
              at the port is exclusive OR'ed with the integer  
              expression J, and then AND'ed with I. If the  
              result is zero, BASIC loops back and reads the  
              data at the port again. If the result is  
              nonzero, execution continues with the next  
              statement. If J is omitted, it is assumed to be  
              zero

CAUTION:     It is possible to enter an infinite loop with  
              the WAIT statement, in which case it will be  
              necessary to manually restart the machine.

Example:     100 WAIT 32,2

2.65 WHILE...WEND

Format:     WHILE <expression>  
             .  
             .  
             [<loop statements>]  
             .  
             .  
             WEND

Purpose:     To execute a series of statements in a loop as long as a given condition is true.

Remarks:   If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example:    90 ^BUBBLE SORT ARRAY A\$  
             100 FLIPS=1 ^FORCE ONE PASS THRU LOOP  
             110 WHILE FLIPS  
             115     FLIPS=0  
             120     FOR I=1 TO J-1  
             130         IF A\$(I)>A\$(I+1) THEN  
                              SWAP A\$(I),A\$(I+1):FLIPS=1  
             140     NEXT I  
             150 WEND

2.66 WIDTH

Format:       WIDTH [LPRINT] <integer expression>

Purpose:       To set the printed line width in number of characters for the terminal or line printer.

Remarks:     If the LPRINT option is omitted, the line width is set at the terminal. If LPRINT is included, the line width is set at the line printer.

<integer expression> must have a value in the range 15 to 255. The default width is 72 characters.

If <integer expression> is 255, the line width is "infinite," that is, BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

Example:      10 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  
              RUN  
              ABCDEFGHIJKLMNOPQRSTUVWXYZ  
              Ok  
              WIDTH 18  
              Ok  
              RUN  
              ABCDEFGHIJKLMNOPQR  
              STUVWXYZ  
              Ok

## 2.67 WRITE

Format:      WRITE[<list of expressions>]

Purpose:      To output data at the terminal.

Remarks:    If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output at the terminal. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement, Section 2.49.

Example:     10 A=80:B=90:C\$="THAT'S ALL"  
             20 WRITE A,B,C\$  
             RUN  
             80, 90,"THAT'S ALL"  
             Ok

2.68 WRITE#

Format: WRITE#<file number>,<list of expressions>

Purpose: To write data to a sequential file.

Remarks: <file number> is the number under which the file was OPENed in "O" mode. The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

Example: Let A\$="CAMERA" and B\$="93604-1". The statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

would input "CAMERA" to A\$ and "93604-1" to B\$.